Badger - A Fast and Provably Secure MAC

Martin Boesgaard, Ove Scavenius, Thomas Pedersen, Thomas Christensen, and Erik Zenner

CRYPTICO A/S
Fruebjergvej 3
2100 Copenhagen
Denmark
info@cryptico.com

Abstract. We present Badger, a new fast and provably secure MAC based on universal hashing. In the construction, a modified tree hash that is more efficient than standard tree hash is used and its security is being proven. Furthermore, in order to derive the core hash function of the tree, we use a novel technique for reducing Δ-universal function families to universal families. The resulting MAC is very efficient on standard platforms both for short and long messages. As an example, for a 64-bit tag, it achieves performances up to 2.2 and 1.3 clock cycles per byte on a Pentium III and Pentium 4 processor, respectively. The forgery probability is at most $2^{-52.2}$.

Keywords. MAC, universal hash, tree, pseudo-random generator

1 Introduction

A Message Authentication Code (MAC) provides a way to detect whether a message has been tampered with during transmission. The usual model for authentication includes three participants: a transmitter, a receiver and an opponent. The transmitter sends a message over an insecure channel, where the opponent can introduce new messages as well as alter existing ones. Insertion of a new message by the opponent is called impersonation, and modification of an existing message by the opponent is called substitution. In both cases the opponent's goal is to deceive the receiver into believing that the new message is authentic.

In many applications, it is of significant importance that the receiver can verify the integrity of a message. In some cases this is even more important than encryption [12]. Often encryption and authentication are both required. With the emergence of fast software-based encryption algorithms like Rijndael [8], SNOW [10], Rabbit [6] etc., the need for fast software-based message authentication codes is increasing. Some attempts have been made to construct integrated MAC and encryption algorithms, e.g. Helix [13]. However, such approaches make it hard to prove the security of the MAC part. In contrast, MACs that can be proven secure with respect to an underlying cryptographic primitive exist. Prominent examples are HMAC [16] and the universal hashing approach [7].

The construction presented here is based on the universal hashing paradigm introduced by Carter and Wegman [7, 23]. They proposed to hash a given message with a randomly chosen function from a strongly universal family of hash functions, whereafter the output is encrypted with a one-time-pad (OTP) in order to obtain the MAC tag. Since universal hash functions are only required to fulfill, in a cryptographical sense, a rather simple combinatorial property, they can usually be constructed to be very fast. Recent research

has been successful in achieving high speed for long messages. Notable examples can be found in [19, 3, 11, 4, 14]. However, for short messages, these algorithms lose some of their efficiency due to initialization and finalization overhead; a problem that was addressed e.g. by new versions of UMAC [17].

It is the aim of this paper to construct a Wegman-Carter based MAC which is fast on both short and long messages. The performance on short messages is important, as e.g. the MAC function used in IPsec operates on 43-1500 bytes (see chapter 3 of [17]) and the MAC function used in TLS operates on 0-17 kilobytes. In addition, the setup procedure must be simple and fast, as the number of messages and amount of data processed per setup is small in many applications, e.g. TLS. Finally, the MAC should provide verifier-selectable assurance[1].

In order to achieve high performance we introduce new families of universal hash functions especially well suited for tree-like hashing. These are obtained using a novel technique for reducing Δ-universal hash families to universal hash families. This results in significant performance gains for small compressions. Furthermore, we develop an effective tree-like hashing procedure which basically consists of combining a tree hash with a linear hash. The construction is provably secure (relative to a cryptographic primitive) with simple proofs.

Organization: The paper is organized as follows. In section 2 we present the definitions of the different classes of universal hash families, we review composition theorems and sketch our construction. In section 3 we introduce a simple method to reduce Δ-universal hash families to universal hash families. A modification of the standard tree hashing scheme is presented in section 4. Section 5 discusses how to build a strongly universal hash family from this scheme. Section 6 contains the specification of Badger, and performance results are presented in section 7. We conclude in section 8.

2 Universal Hashing and Message Authentication

In 1981, Wegman and Carter [23] showed that randomly chosen elements from a strongly universal hash function family can be used to compress a given message and encrypt the output using a OTP[2]. We describe briefly in the following why this is possible, and how it will be used in our design.

Universal hash function families: The following definitions of universal hash function families are well-known from the literature.

Definition 1. *[7, 20] An ϵ-almost universal (ϵ-AU) family H of hash functions maps from a set A to a set B, such that for any distinct elements $a, a' \in A$:*

$$\Pr_{h \in H}[h(a) = h(a')] \le \epsilon \qquad (1)$$

H is universal (U) if $\epsilon = 1/|B|$.

[1] For a more detailed description of verifier-selectable assurance, see [17]. In short, this means that the receiver can choose to verify to lower assurance levels than for the full tag in order to increase performance.

[2] Of course, a cryptographic pseudo-random generator (PRG) can also be used to generate a pseudo-random pad, but then the security depends on the security of the PRG.

Definition 2. *[15, 22] Let $(B, +)$ be an Abelian group. A family H of hash functions that maps from a set A to the set B is said to be ϵ-almost Δ-universal (ϵ-AΔU) w.r.t. $(B, +)$, if for any distinct elements $a, a' \in A$ and for all $\delta \in B$:*

$$\Pr_{h \in H}[h(a) - h(a') = \delta] \leq \epsilon \tag{2}$$

H is Δ-universal (ΔU) if $\epsilon = 1/|B|$.

Definition 3. *[23, 20] An ϵ-almost strongly-universal (ϵ-ASU) family H of hash functions maps from a set A to a set B, such that for any distinct elements $a, a' \in A$ and all $b, b' \in B$:*

$$\Pr_{h \in H}[h(a) = b] = 1/|B| \qquad \text{and} \tag{3}$$

$$\Pr_{h \in H}[h(a) = b, h(a') = b'] \leq \epsilon/|B| \tag{4}$$

H is strongly universal (SU) if $\epsilon = 1/|B|$.

The Wegman-Carter MAC: From the definitions it follows that strongly universal hashing can be used for message authentication. If we denote the probability for an impersonation attack to succeed by P_i and the probability for a substitution attack to succeed by P_s, we have the following theorem:

Theorem 1. *[23, 21, 18] There exists an ϵ-ASU family of hash functions from A to B if and only if there exists an authentication code with $|A|$ messages, $|B|$ authenticators and $k = |H|$ keys, such that $P_i = 1/|B|$ and $P_s \leq \epsilon$.*

The particular Wegman-Carter MAC can be defined as follows:

Definition 4. *Given an ϵ-ASU family $\mathcal{H}$ of hash functions mapping from a set A to a set B, a nonce n, and an OTP $r(n)$, then the Wegman-Carter* MAC *is*

$$\mathrm{MAC}_{\mathrm{WC}}(M; h, r(n)) = h(M) \oplus r(n), \tag{5}$$

where h is a random hash function from $\mathcal{H}$ and M is the message.

A new nonce must be used for each application of the MAC to ensure the unconditional security of the construction.

Composition rules: Hash families can be combined in order to obtain new hash families. The below composition rules (see [21]) describe what happens to the resulting ϵ, domains, and ranges.

Composition 1 *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-AU family H_2 of hash functions from B to C, then there exists an ϵ-AU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2$.*

Composition 2 *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-ASU family H_2 of hash functions from B to C, then there exists an ϵ-ASU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2$.*

Our construction: In the following, we will use composition rule 2 to construct a Wegman-Carter MAC. First, we will use an ϵ_{H^*}-AU universal function family H^* to hash messages of all sizes onto a fixed size. Subsequently, we will use an ϵ_F-ASU function family F to guarantee for the strong universality of the overall construction. Thus, the strongly universal hash family used for our MAC can be described as $\mathcal{H} = H^* \times F$. Note that the following theorem follows immediately from composition rule 2:

Theorem 2. *The hash function family $\mathcal{H} = H^* \times F$ is $\epsilon_F + (1 - \epsilon_F)\epsilon_{H^*}$-ASU.*

We proceed by describing H^* in sections 3 and 4 and F in section 5.

3 Reducing AΔU Families to AU Families

Reducing function families: Note that for the classes of hash function families defined in definitions 1-3, the latter are contained in the former, i.e. an AΔU family is also an AU family a.s.o. On the other hand, a stronger family can be reduced to a weaker one. This is, of course, only relevant when a performance gain can be achieved. In the following, we will describe a method to reduce Δ-universal hash functions to universal hash functions. It turns out that these new universal hash families are particularly well-suited for tree structures.

Theorem 3. *Let H^{Δ} be an ϵ-AΔU hash family from a set A to a set B. Consider a message $(m, m_b) \in A \times B$. Then the family H consisting of the functions $h(m, m_b) = h^{\Delta}(m) + m_b$ is ϵ-AU.*

Proof. From the definitions above we have

$$\begin{aligned}\Pr_{h \in H}[h(m, m_b) - h(m', m'_b) = 0] &= \Pr_{h^{\Delta} \in H^{\Delta}}[h^{\Delta}(m) + m_b - h^{\Delta}(m') - m'_b = 0] \\ &= \Pr_{h^{\Delta} \in H^{\Delta}}[h^{\Delta}(m) - h^{\Delta}(m') = m'_b - m_b].\end{aligned}$$

If $m \neq m'$, then this probability is at most ϵ, since H^{Δ} is an ϵ-AΔU family. If $m = m'$ but $m_b \neq m'_b$, then the probability is trivially 0. □

Constructing the ENH family: A very fast universal hash family is the NH family used in UMAC [17]:

$$\mathrm{NH}_K(M) = \sum_{i=1}^{l/2} (k_{2i-1} +_w m_{2i-1}) \cdot (k_{2i} +_w m_{2i}) \bmod 2^{2w}, \tag{6}$$

where '$+_w$' means 'addition modulo 2^w', and $m_i, k_i \in \{0, ..., 2^w - 1\}$. It is a 2^{-w}-AΔU hash family. In [17], the AΔU property is mentioned, but only the AU property is explicitly proven.

Lemma 1. *The following version of NH is 2^{-w}-AΔU:*

$$\mathrm{NH}_K(M) = (k_1 +_w m_1) \cdot (k_2 +_w m_2) \bmod 2^{2w}. \tag{7}$$

Proof. This proof is just a slight modification of the one presented in [17]. We must show that

$$\Pr_{k_1,k_2}[(k_1 +_w m_1)(k_2 +_w m_2) - (k_1 +_w m_1')(k_2 +_w m_2') = \delta] \leq 2^{-w}.$$

where all arithmetic is carried out modulo 2^{2w}. Assume that $m_2 \neq m_2'$. Define $c = k_2 + m_2$ and $c' = k_2 + m_2'$. By assumption it follows that $c \neq c'$. So we have

$$\Pr_{k_1,k_2}[(k_1 +_w m_1)c - (k_1 +_w m_1')c' - \delta = 0] \leq 2^{-w}.$$

since from lemma 2.4.3 in [17], the equality will only be satisfied by one k_1. □

Choosing $w = 32$ and applying theorem 3, we obtain the 2^{-32}-AU function family ENH, which will be the basic building block of our MAC:

$$\begin{aligned} &\mathrm{ENH}_{k_1,k_2}(m_1, m_2, m_3, m_4) \\ &\quad = (m_1 +_{32} k_1)(m_2 +_{32} k_2) +_{64} m_3 +_{64} 2^{32} m_4, \end{aligned} \tag{8}$$

where all arguments are 32-bit and the output is 64-bit.

4 The Modified Tree Construction

The standard tree construction: The ENH function family maps 128-bit inputs to 64-bit outputs. An immediate use of such a function is in a tree-like structure that allows hashing of messages of arbitrary length. More generally, assume a block length b, a universal hash family H that maps from bc to b bits, and a message of length $|M| = b \cdot c^n$, for some suitable value n. Let $m||m'$ denote the concatenation of two strings m, m', and let $f \circ f'$ denote the successive execution of function f' and f. Then a hash tree can be defined by a succession of n parallel hashes, as follows [7, 1]:

Definition 5. *Let H be a universal hash family, taking bc bits to b bits. Given a message $M = m_1||...||m_{c^n}$ with length $|M| = bc^n$, we hash c blocks at a time with a function $h \in H$ and concatenate the results. The result is a string of length bc^{n-1}. We denote the hash family by H^{par} and a member by h^{par}.*

$$h^{\mathrm{par}}(M) = h(m_1, ..., m_c)||...||h(m_{c^n-c+1}, ..., m_{c^n}) \tag{9}$$

It is easy to see that if H has a collision bound of ϵ then so does the parallel hash, H^{par}. We define the standard tree construction as follows:

Definition 6. *Let M and H be as in definition 5. We define a new hash family by applying h_i^{par} n times, each time with a new random $h_i \in H$. We denote the resulting function family by H_n^{tree} and a member by h_n^{tree}:*

$$h_n^{\mathrm{tree}}(M) = h_n^{\mathrm{par}} \circ h_{n-1}^{\mathrm{par}} \circ ... \circ h_1^{\mathrm{par}}(M).$$

Theorem 4. *The function family H_n^{tree} is a $1-(1-\epsilon)^n$-universal family of hash functions for equal length messages.*

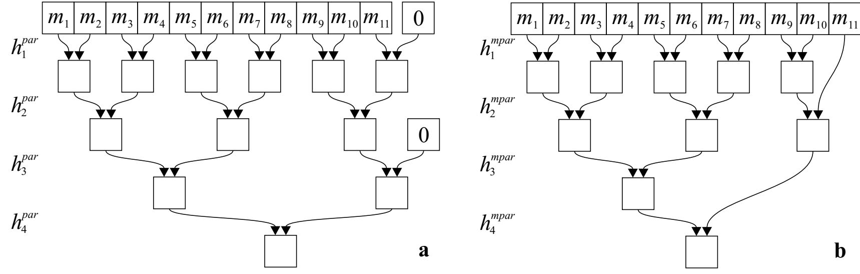

Fig. 1. Figure (a) illustrates the standard tree construction using the parallel hash and figure (b) illustrates the modified tree construction using the modified parallel hash.

Proof. Let us define ϵ_i as the collision bound for H_i^{tree}, then we have for H_{i+1}^{tree}:

$$\Pr[h_{i+1}^{\text{par}}(h_i^{\text{tree}}(m)) - h_{i+1}^{\text{par}}(h_i^{\text{tree}}(m')) = 0] \leq \epsilon_i(1-\epsilon) + \epsilon.$$

Solving the recurrence we get:

$$\begin{aligned}\Pr[h_n^{\text{par}}(h_{n-1}^{\text{tree}}(m)) - h_n^{\text{par}}(h_{n-1}^{\text{tree}}(m')) = 0]& \\ \leq (1-\epsilon)^{n-1}\epsilon + \epsilon \sum_{i=1}^{n-2}(1-\epsilon)^i + \epsilon& \\ = 1-(1-\epsilon)^n&\end{aligned}$$

□

The modified tree construction: Consider, as an example, the case $c = 2$, yielding a binary tree. Then the message length must be $b \cdot 2^n$, for some suitable n. If that is not the case, Wegman and Carter propose [23] to break the message into substrings of length $2b$ and if necessary pad the last substring with zeroes. The resulting string is hashed with the parallel hash. If necessary, the resulting string is again padded with zeroes. This is repeated until the resulting string has length b. This procedure is illustrated in fig. 1a.

Note that this algorithm is not always optimal, because for message lengths not equal to a power of two, extra applications of the universal hash function are required. Of course, this is only significant for short messages. We start constructing a modified tree hash by defining a modified parallel hash, as follows:

Definition 7. *Given a universal hash family, H, whose members h take bc bits to b bits, consider the message $M = m_1||..||m_q$ where $|M| = bq$. Let $r = q \bmod c$, then the modified parallel hash can be defined as:*

$$h^{\text{mpar}}(M) = \begin{cases} h(m_1,..,m_c)||..||h(m_{q-c+1},..,m_q) \text{ if } r = 0 \\ h(m_1,..,m_c)||..||h(m_{q-c-r+1},..,m_{q-r})||m_{q-r+1}||..||m_q \text{ if } r \neq 0 \end{cases} \tag{10}$$

Lemma 2. *The modified parallel hash is ϵ-AU on equal length messages.*

Proof. In the first case, where q is a multiple of c we simply have a parallel hash and the bound on the collision probability is ϵ. In the case where q is not a multiple of c, there are two possible situations. Either the difference in the messages M and M' is in the part, which is processed by h, or in the part which is not processed but simply concatenated to the result. In the first situation the bound on the collision probability is ϵ. In the second situation the collision probability is trivially zero. Thus, the collision probability is at most ϵ. □

It is straightforward to define a modified tree hash, i.e. define it as in Definition 6 but use the modified parallel hash instead of the usual parallel hash.

Corollary 1. *Given a message with length $|M| = bq$, where $c^{n-1} < q \leq c^n$, the modified tree hash defines a $1-(1-\epsilon)^n$-AU family of hash functions on equal length messages.*

Proof. This follows from theorem 4, when the usual parallel hash is replaced by the modified parallel hash, since both are ϵ-AU, and the number of levels is the same in both cases[3]. □

The binary case: Again, consider $c = 2$. The message is divided into blocks of size b. If the message length is not a multiple of b, zeros are appended to the message such that the length becomes a multiple of b. If the length hereafter is a multiple of $2b$, the hash function is applied to each block and the results are concatenated. If the length is an odd multiple of b, the hash function is applied to each block except the last block. The results and the last block are concatenated. The procedure is repeated until the size of the result is b. The construction is illustrated in Fig. 1b.

A different view: Note that the construction can be defined in an alternative way. Considering again the case $c = 2$, the message length can be described by $|M| \equiv b\sum_{i=0}^{n} a_i 2^i$ with $a_i \in \{0, 1\}$. To each term $a_i = 1$ in the sum there corresponds a tree with i levels. We order these trees according to size with the largest tree first. More precisely, we use the tree hash for each group of data corresponding to a term in the sum, concatenate the result, and linearly hash it backwards, i.e. take the b-bit block as output from the last tree and hash it with the result of the second to last tree and so on, until only one b-bit string is left. In other words, the construction consists of a series of concatenated tree hashes followed by a linear hash [1]. For the example in Fig. 1b, the message length can be written as: $|M| = b(2^3 + 2^1 + 2^0)$. There is one tree with 3 levels, one with 1 level and one with 0 levels. The hash results if the outputs of those trees are linearly hashed starting with the result from the smallest tree.

The function family H^*: The above construction is only AU for equal length messages. To ensure universality for different length messages, we simply concatenate the length of the given message in a fixed z-bit format [17, 1]:

[3] In a Wegman-Carter binary tree hash, a message consisting of an odd number of blocks is padded such that the number of blocks is even. This is done after each application of the parallel hash. The number of levels is equal to the number of levels for a message whose length is the nearest larger power of two. Now it is easy to convince oneself that the number of levels of the modified tree hash is exactly the same.

Definition 8. *Fix $z > 0$ and let the message M (before padding) have any length less than 2^z. Define $L_z = |M|$ to be the z-bit representation of the length and define the family H^* by its members h^*, as follows:*

$$h^*(M) = L_z || h(M). \tag{11}$$

We then have the following property:

Lemma 3. *The hash function family H^* is $1 - (1 - \epsilon)^n$-AU.*

Proof. In the case $|M| \neq |M'|$, the collision probability is trivially zero. In the case $|M| = |M'|$, the collision probability is defined according to corollary 1 by the number of levels necessary to compress the message. □

Note that for Badger, we will use H = ENH. This immediately yields a binary tree with a block size of 64 bit. The input size of the function family H^* is defined to be between 0 and $2^{64} - 1$ bit. Consequently, the output size is 128 bit, 64 bit each for the hash and for the message length. Also note that the tree will contain between 1 and 58 levels, yielding a collision probability between $\epsilon_{H^*} \leq 2^{-32}$ for small and $\epsilon_{H^*} = 1 - (1 - 2^{-32})^{58} \leq 2^{-26.14}$ for large trees.

5 The function family F

Strenghtening a function family: What is left now according to theorem 1 is to construct a suitable SU family F, such that the overall function family $\mathcal{H} = H^* \times F$ is both efficient and secure. Note that without considering the details of H^*, the input size of F is $b + z$ bit, and its output size should be equivalent to the security of the overall scheme.

In section 3, we reduced a strong class of universal hash functions to a weaker one. In order to construct the strongly universal function familiy F, we will do the opposite: In accordance with lemma 1 from [11], we will transform the Δ-universal hash family, MMH*, proposed by Halevi and Krawczyk [14] based on [7], into a strongly universal hash family. This is accomplished by adding an additional key, k_{l+1}, in the following way:

$$\mathrm{MMH}^{\mathrm{su}}_K(M) = \sum_{i=1}^{l} m_i k_i + k_{l+1} \bmod p, \tag{12}$$

where p is a prime number, $M = m_1||...||m_n$, and $m_i, k_i \in \{0, ..., p-1\}$.

Key material: It is easily seen that for a given message M, the amount of key material, $N_{\mathcal{H}}(M)$, needed to choose a function from the family $\mathcal{H}$ is defined by

$$N_{\mathcal{H}}(M) = N_H \lceil \log_c(|M|/b) \rceil + N_F, \tag{13}$$

where N_H is the amount of key material needed for the H-function in the tree and N_F is the amount of key material needed for the F-function. Note that the amount of key material required is the same as for the usual tree MAC.

The choice for Badger: Remember from section 4 that H^* produces a 128-bit output, which is also the minimum input size for F. Also remember that the collision probability for the H^*-function ranges from 2^{-32} to $2^{-26.14}$, depending on the size of the tree. Since the overall security can not get better than that (according to theorem 2), an output size of 32 bit for the F-function is sufficient, since additional bits do not improve the security.

Consequently, we use a 32-bit version of the MMH$^{\mathrm{su}}$-construction. We take p to be the largest 32-bit prime number, which is $p = 2^{32} - 5$. In order to process a 128-bit input, we have to choose $n = 5$ and obtain:

$$\mathrm{F}_K(M) = \sum_{i=1}^{5} q_i k_i + k_6 \bmod \left(2^{32} - 5\right). \tag{14}$$

Note that the 128-bit output of H^* has to be divided into five input blocks q_i in some way. For Badger, it is padded with 7 leading zeroes and then split into 27-bit blocks[4]. The rationale for this design and an efficient way to implement it is given in section A in the appendix. In section B of the appendix, a mathematical simplification of the F-function is discussed, along with an explanation why it is not used for Badger.

6 The Badger Specification

For the algorithmic description of Badger, the following pseudocode calls to external functions are made:

- `PRG_KeySetup(`K`)`: Initializes pseudorandom generator with the 128-bit key K.
- `PRG_IVSetup(`N`)`: Initializes pseudorandom generator with the 64-bit nonce N.
- `PRG_Nextbit(`n`)`: Returns n bit of pseudorandom output.

Key generation: To generate the key material for the H^*- and F-functions, any secure PRG can be used, as long as the key length is at least 128 bit. Since the height of the tree is 58 for maximum size messages, we require 58 64-bit keys for the H^*-function. Furthermore, 6 keys from the interval $\{0, ..., 2^{32} - 6\}$ have to be generated for the F-function. Note that (as opposed to the key material for the pseudo-random pad), this key material can be computed once and then be re-used for the computation of all future MACs. The full procedure is given in figure 2.

Message processing: Pseudo-code for Badger is presented in figure 3. To process the message, it is first divided into 64-bit blocks and padded with zeroes if necessary. The resulting bit string is then compressed into one 64-bit block, using the H^*-function. The length of the message in bit (before padding) is represented as a 64-bit number and concatenated to the 64-bit result. The resulting 128-bit block is prefixed with 7 zeroes, divided into five 27-bit blocks and run through the F-function.

The final tag is generated by xor-ing the output of the hash function with a pseudo-random pad, according to Definition 4. Note that no output of the PRG must ever be re-used; this can be achieved by running the PRG without resetting, or by using a new nonce for every new message.

[4] Note that the security claims for the SU function family also hold if not all messages from $\{0, ..., 2^{32} - 6\}$ are actually used as inputs, as long as all keys from $\{0, ..., 2^{32} - 6\}$ occur with equal probability.

```
PRG_KeySetup(K)
words_used = 0

// Assign 32-bit values to finalize keys
for j = 1 to 6:
   for i = 1 to u:
      final_key[j][i] = PRG_Nextbit(32)
      words_used++

// Test whether they are in Z_p
for j = 1 to 6:
   for i = 1 to u:
      while(final_key[j][i] ≥ p)
         final_key[j][i] = PRG_Nextbit(32)
         words_used++

// Empty buffer
while(words_used mod 4 ≠ 0):
   discard PRG_Nextbit(32)
   words_used++

// Assign 64-bit values to level keys
for j = 1 to v:
   for i = 1 to u:
      level_key[j][i] = PRG_Nextbit(64)
```

Fig. 2. Pseudo-code of the key setup

Forgery probability: The forgery probability is $\epsilon \leq \epsilon_F + (1 - \epsilon_F)\epsilon_{H^*}$, according to theorem 2. Remember that depending on the message length, the upper bound on ϵ_{H^*} can range from 2^{-32} to $2^{-26.14}$. Also note that $\epsilon_F = 1/(2^{32} - 5) \approx 2^{-32}$. Using these values, it can be seen that the overall forgery probability has an upper bound ranging from 2^{-31} for extremely short to $2^{-26.12}$ for extremely long messages.

Forgery probabilities of up to $2^{-26.12}$ are insufficient for most applications. However, a simple method to reduce the forgery probability is to hash the message u times with independent keys and concatenate the results. This results in a forgery probability of ϵ^u. To obtain 128-bit security, we need to hash the message 5 times, yielding bounds on the forgery probability of between 2^{-155} and $2^{-130.6}$ and a tag size of 160 bits. In particular, this leads to the verifier-selectable assurance as each 32-bit tag can be verified independently.[5]

[5] At first glance, it seems that the Toeplitz construction (proposed by Krawczyk in [15]) is applicable here, i.e. that the u parallel MACs are calculated using $(k_1, \ldots, k_{58})$, $(k_2, \ldots, k_{59})$ etc. However, this only makes sense if the resulting forgery probability is at most ϵ^u, and experiments with smaller versions of the H^*-function indicate that this is not the case here. Thus, the Toeplitz construction is not used with Badger.

```
Function h(k, m_1, m_2)
1. return (m_1 +_32 k) · ((m_1 ≫ 32) +_32 (k ≫ 32)) +_64 m_2

Function Keygen(k)
1. generate k_1^F, ..., k_6^F ∈ {0, ..., 2^32 − 6} from PRG(k)
2. generate k_1, ..., k_58 ∈ {0, ..., 2^32 − 1} from PRG(k)
3. return k_1^F, ..., k_6^F, k_1, ..., k_58

Function Badger(k, M, N)
1. L = |M|
2. while |M| mod 64 ≠ 0 do: M = M||0
3. for i = 1 to i = ⌈log_2(L/64)⌉ do:
4.    divide M into 64-bit blocks, M = m_1||...||m_t
5.    if t is even:
         M = h(k_i, m_1, m_2)||...||h(k_i, m_{t−1}, m_t)
      else:
         M = h(k_i, m_1, m_2)||...||h(k_i, m_{t−2}, m_{t−1})||m_t
6. set Q = 0^7||L||M
7. divide Q into 27-bit blocks , Q = q_5||...||q_1
8. S = Σ_{i=1}^{5} q_i k_i^F + k_6^F mod (2^32 − 5)
9. return S ⊕ PRG(k, N)
```

Fig. 3. Pseudo-code of the Badger algorithm

7 Performance

On the testing environment: Performance of the Badger algorithm was measured on a 1.0 GHz Pentium III and on a 1.7 GHz Pentium 4 processor. The speed-optimized versions were programmed in assembly language inlined in C and compiled using the Intel C++ 7.1 compiler. All performance results in this section are based on generating a $2 \cdot 32$ bit tag. The pseudo-random material required for the algorithm was generated using the stream cipher Rabbit [6, 5], which is very fast in software. Note that since Badger is designed with speed being a main objective, it makes sense to use a fast stream cipher (instead of, e.g., using a block cipher like AES in a suitable stream cipher mode).

On IV-setup: Note that the pseudo-random pad can be generated either with or without an explicit IV-setup. If an explicit IV is used, the stream cipher has to be re-initialized for each message. Without an explicit IV, the key material for successive messages is produced by continuous extraction of bytes from the stream cipher, yielding a performance advantage. This corresponds to interpreting the message number as the IV. However, this technique is only applicable if messages are guaranteed to be received in the same order as generated, which is often not the case (e.g. in IPsec communication). Table 1 gives performance numbers both with and without explicit IV-setup.

On short messages: Since the amount of key material required for Badger depends on the length of the message, optimized versions can be used in applications where the message length is upper bounded. For example, in typical IPsec applications, the message length cannot exceed 1500 bytes and when authenticating TLS protected data, each message

Table 1. Performance results with and without IV-setup. "Key setup" generates all keys for the ϵ-AU and SU hash functions, "Universal hash" processes the tree, and "Finalization" includes the F-function and generates the pseudo-random pad.

Function	Pentium III	Pentium 4
Key setup	4093 cycles	5942 cycles
Universal hash	2.2 cycles/byte	1.3 cycles/byte
Finalization without IV	175 cycles	220 cycles
Finalization with IV	433 cycles	892 cycles

Table 2. Badger properties for various restricted message lengths. "Memory req." denotes the amount of memory required to store the internal state including key material and the inner state of the Rabbit stream cipher. "Setup" denotes the key setup, and "Fin." denotes finalization with IV-setup.

Max.	Forgery	Memory	Pentium III		Pentium 4	
message size	prob.	req.	Setup	Fin.	Setup	Fin.
2^{11} bytes (e.g. IPsec)	$2^{-57.7}$	400 bytes	1133 cycles	409 cycles	1862 cycles	868 cycles
2^{15} bytes (e.g. TLS)	$2^{-56.6}$	528 bytes	1370 cycles	421 cycles	2190 cycles	880 cycles
2^{32} bytes	$2^{-54.2}$	1072 bytes	2376 cycles	421 cycles	3576 cycles	880 cycles
$2^{61}-1$ bytes	$2^{-52.2}$	2000 bytes	4093 cycles	433 cycles	5942 cycles	892 cycles

cannot exceed 17 kilobytes [9]. Furthermore, the evaluation of the F-function is simplified since part of the input is zero, see eq. (12). The properties of Badger when the message length is limited are shown in Table 2.

Note that the performance numbers for the key setup and finalization (which are dependent on the PRG in use) are partially based on estimates. The numbers for the universal hash function, however, are independent of the PRG and are fully based on measurements.

On comparing performance: It has become common practice in scientific papers proposing new cryptographic primitives to provide performance comparisons with competing designs. However, there are numerous problems associated with such comparisons. For example, in the case of MACs, the performance figures given would be influenced by, e.g.,

- the amount of optimization done on the algorithm implementation,
- the exact choice of the message lengths under consideration[6],
- the choice of the underlying cryptographic primitive (like Rabbit, AES etc.),
- the amount of optimization done on the implementation of this primitive,
- the amount of memory used to optimize, e.g. by using lookup tables, and
- the processor the performance tests have been conducted on.

When using performance numbers provided by others, it is often not clear how at least some of the above parameters have been chosen. On the other hand, when implementing the competitor's algorithm oneself, it is quite likely that one would (even if inadvertedly)

[6] Many algorithms have criticals lengths where the performance suddenly decreases dramatically by increasing the message length by just one bit. This is simply due to the fact that computers have limited register and cache sizes and that by exceeding those sizes by just one bit, things suddenly get more complicated. For an example of such critical message sizes, see e.g. the performance tables of the POLY-1305 MAC given in appendix B of [2]. By deliberately choosing the message lengths in one's own favour, the outcome of a performance comparison could be severely influenced.

optimize one's own brainchild better than the competitor's. Thus, we conclude that a fair performance comparison can only be based on the implementation by an independent entity, and will thus refrain from giving such comparisons here.

8 Conclusion

We presented a new fast and provably secure MAC called Badger, based on universal hashing. In the construction, a modified tree hash was introduced that basically combines a tree hash with a linear hash. The modified tree hash is more efficient than the standard tree hash, and its security has being proven. Furthermore, in order to derive the core hash function of the tree, we introduced a novel technique for reducing Δ-universal function families to universal families. The resulting MAC is very efficient on standard processors both for short and long messages. As an example, for a 64-bit tag, it achieves performances of up to 2.2 and 1.3 clock cycles per byte on a Pentium III and Pentium 4 processor, respectively. The key material necessary for the hash functions is only 976 bytes, and the forgery probability is at most $2^{-52.2}$.

References

1. M. Bellare and P. Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In B. Kaliski, editor, *Proc. Crypto '97*, volume 1294 of *LNCS*, pages 470–484. Springer, 1997.
2. D. Bernstein. The Poly1305-AES message-authentication code. In *Proc. Fast Software Encryption '05*.
3. J. Bierbrauer, T. Johansson, G. Kabatianskii, and B. Smeets. On families of hash functions via geometric codes and concatenation. In D. Stinson, editor, *Proc. Crypto '93*, volume 773 of *LNCS*, pages 331–342. Springer, 1994.
4. J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In M. Wiener, editor, *Proc. Crypto '99*, volume 1666 of *LNCS*, pages 216–232. Springer, 1999.
5. M. Boesgaard, T. Pedersen, M. Vesterager, and E. Zenner. The Rabbit stream cipher - design and security analysis. In *Workshop Record of the State of the Arts of Stream Ciphers Workshop*, pages 7–29. ECRYPT Network of Excellence in Cryptography, October 2004.
6. M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius. Rabbit: A new high-performance stream cipher. In T. Johansson, editor, *Proc. Fast Software Encryption 2003*, volume 2887 of *LNCS*, pages 307–329. Springer, 2003.
7. J. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.
8. J. Daemen and V. Rijmen. AES proposal: Rijndael.
http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf, 1999.
9. T. Dierks and C. Allen. The TLS protocol version 1.0, IETF RFC 2246. http://www.ietf.org/rfc.html, 1999.
10. P. Ekdahl and T. Johansson. A new version of the stream cipher SNOW. In H. Heys and K. Nyberg, editors, *Proc. SAC 2002*, volume 2595 of *LNCS*, pages 47–61. Springer, 2002.
11. M. Etzel, S. Patel, and Z. Ramzan. Square Hash: Fast message authentication via optimized universal hash functions. In M. Wiener, editor, *Proc. Crypto '99*, volume 1666 of *LNCS*, pages 234–251. Springer, 1999.
12. N. Ferguson and B. Schneier. *Practical Cryptography.* Wiley, 2003.
13. N. Ferguson, D. Whiting, B. Schneier, J. Kelsey, S. Lucks, and T. Kohno. Helix: Fast encryption and authentication in a single cryptographic primitive. In T. Johansson, editor, *Proc. Fast Software Encryption 2003*, volume 2887 of *LNCS*, pages 330–346. Springer, 2003.
14. S. Halevi and H. Krawczyk. MMH: Software message authentication in the Gbit/second rates. In E. Biham, editor, *Proc. Fast Software Encryption '97*, volume 1267 of *LNCS*, pages 172–189. Springer, 1997.
15. H. Krawczyk. LFSR-based hashing and authentication. In Y. Desmedt, editor, *Proc. Crypto '94*, volume 839 of *LNCS*, pages 129–139, Berlin, 1994.
16. H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication IETF RFC 2104. http://www.ietf.org/rfc.html, 1997.
17. T. Krovetz. *Software-Optimized Universal Hashing and Message Authentication.* PhD thesis, UC Davis, September 2000.
18. W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In J. Stern, editor, *Proc. Eurocrypt '99*, volume 1592 of *LNCS*, pages 24–41. Springer, 1999.
19. P. Rogaway. Bucket hashing and its application to fast message authentication. In D. Coppersmith, editor, *Proc. Crypto '95*, volume 963 of *LNCS*, pages 29–42. Springer, 1995.
20. D. Stinson. Universal hashing and authentication codes. In J. Feigenbaum, editor, *Proc. Crypto '91*, volume 576 of *LNCS*, pages 74–85. Springer, 1992.
21. D. Stinson. Universal hashing and message authentication codes. *Designs, Codes, and Cryptography*, 4(4):369–380, 1994.
22. D. Stinson. On the connection between universal hashing, combinatorial designs and error-correcting codes. In *Proc. Congressus Numerantium 114*, pages 7–27, 1996.
23. M. Wegmann and J. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

```
r = 5 * a;
r += b;

if (r < b)
  r -= 0xFFFFFFFB;        // if overflow has occured
else if (r >= 0xFFFFFFFB)
  r -= 0xFFFFFFFB;        // if result ≥ 2^32 − 5
```

Fig. 4. Computing modulus $2^{32} - 5$ in the F-function

A Efficient computation of the F-function

Computation of the modular reduction: Remember from section 5 that the F-function is of the form

$$\mathrm{F}_K(M) = \underbrace{\sum_{i=1}^{5} q_i k_i + k_6}_{t} \bmod \left(2^{32} - 5\right),$$

where a 128-bit input has to be divided into five variables $q_1, \ldots, q_5$. Assuming that additions of 64-bit variables and multiplications of 32-bit values can be implemented efficiently on standard processors, the major problem is the calcuation of the modular reduction.

Also remember that the 128-bit output of H^* is padded with 7 leading zeroes and divided into 5 blocks q_i of length 27 bit. This has the effect of guaranteeing that $t < (2^{64} - 1)/5$. In particular, the result of the full addition can be calculated as a 64-bit value without any need to handle of carry bits. All that remains is to calculate $t \bmod \left(2^{32} - 5\right)$.

Here, we use the well-known fact that

$$(a \cdot 2^n + b) \bmod (2^n - k) = (a \cdot k + b) \bmod (2^n - k)\,.$$

In the concrete case, if we write $t = (a||b)$, with a being the upper and b being the lower word, then the output can be computed as $5a + b \bmod \left(2^{32} - 5\right)$. Note that since $t < (2^{64} - 1)/5$, we have $a < (2^{32} - 1)/5$ and thus $5 \cdot a < 2^{32} - 1$. Again, not carry occurs, this time for a 32-bit addition. In fact, the only carry that can occur is when adding the 32-bit words $5a$ and b. If this happens, we subtract $2^{32} - 5$ from the result, as described in figure 4.

Finally, we have to check whether the result is $\geq 2^{32} - 5$. If this is the case, again, we have to subtract $2^{32} - 5$. Close examination reveals that if an addition overflow occurs, then the value of r is in the range $\{0, \ldots, \mathtt{0xC7FFFFE0}\}$, which means that the second condition can not be true. Thus, only one of the conditions can hold.

Saving additions for short messages: For Badger, the input was padded to 135 bits by prefixing zeroes. Thus, the total input to the F-function is of the form $0^7||L||M$, where L is the 64-bit representation of the message length, and M is the output of the H^*-function. Note that if the first 20 bits of L are zero (i.e., $L < 2^{44}$), then the term $q_5 k_5$ is zero and can be left out to increase performance. Likewise, if the first 47 bits of L are zero (i.e., $L < 2^{17}$ bit), then the term $q_4 k_4$ can be left out, too. This possible saving in computing time (both in key generation and in finalizing) is the reason for choosing 27-bit blocks q_i instead of 26-bit blocks, as would have been possible.

B A mathematical simplification of the F-function

MAC from Δ-universal families: An obvious generalization of the Wegman-Carter MAC is as follows:

Definition 9. *Let $(B, \boxplus)$ be an Abelian group, and $\mathcal{H}$ be an ϵ-AΔU function family mapping from a set A to the set B. Using a nonce n and a random pad $r(n)$, the Δ-MAC is defined as*

$$MAC_{\Delta}(M; h, r(n)) = h(M) \boxplus r(n),$$

where h is a random hash function from $\mathcal{H}$ and M is the message.

Then in a fashion similar to the proof on ϵ-AXU function families [15], the following can be shown:

Theorem 5. *The probability of an impersonation attack to succeed against the Δ-MAC is $1/|B|$, and the probability of a substitution attack to succeed is at most ϵ.*

Also note that in a fashion similar to [21], the following composition rule can be proven:

Composition 3 *If there exists an ϵ_1-AU family H_1 of hash functions from A to B and an ϵ_2-AΔU family H_2 of hash functions from B to C, then there exists an ϵ-AΔU family H of hash functions from A to C, where $H = H_1 \times H_2$, $|H| = |H_1| \cdot |H_2|$, and $\epsilon = \epsilon_1 + \epsilon_2 - \epsilon_1\epsilon_2$.*

Thus, the construction presented in the paper could be simplified by using an F-function from an ϵ-AΔU family, as long as its output is combined with the pseudo-random pad by using the group operation $\boxplus$.

Possible simplification 1: Remember that the family MMH* (without the final key addition) is ΔU. Thus, we could also use the following F-function for our construction:

$$F_K^{\Delta}(M) = \sum_{i=1}^{5} q_i k_i \bmod \left(2^{32} - 5\right),$$

i.e. we could leave out the final key addition altogether. In this case, we would have to implement the pseudo-random pad over the group $(\{0, \ldots, 2^{32} - 6\}, \boxplus)$, where $\boxplus$ denotes the addition modulo $2^{32} - 5$. This, however, means that for all k_i and all pseudo-random words from the PRG, it has to be checked whether they are $< 2^{32} - 5$. This introduces more computational overhead than generating and adding one additional key k_6; thus, we chose the solution proposed in section 5.

Possible simplification 2: Another possible use for theorem 5 is to use a function F that is Δ-universal with regards to the group $(\{0,1\}^n, \oplus)$. A possible solution would be to use a version of MMH* that operates over GF(2^{32}). In fact, it can be shown that this version of MMH* is ΔU, too. If $\oplus$ and $\odot$ denote the addition and multiplication in GF(2^{32}), respectively, then we have

$$F_K^{\oplus}(M) = (q_1 \odot k_1) \oplus \ldots \oplus (q_4 \odot k_4).$$

However, this involves four polynomial multiplications, which are much less efficient on standard processors than additions over the integers. Thus, we discarded this solution, too.